This tutorial demonstrates a simple workflow using ksqlDB to write streaming queries against messages in Apache Kafka® in a Docker environment.

To get started, you must start a Kafka cluster, including ZooKeeper, and a Kafka broker. ksqlDB queries messages from this Kafka cluster.

Prerequisites:

- Docker:
  - Docker version 1.11 or later is installed and running [https://docs.docker.com/engine/installation/].
  - Docker Compose is installed [https://docs.docker.com/compose/install/]. Docker Compose is installed by default with Docker for Mac.
  - Docker memory is allocated minimally at 8 GB. When using Docker Desktop for Mac, the default Docker memory allocation is 2 GB. You can change the default allocation to 8 GB in **Docker** > **Preferences** > **Advanced**.
- Git [https://git-scm.com/downloads].
- Internet connectivity.
- Ensure you are on an Operating System [https://docs.confluent.io/current/installation/versions-interoperability.html#operating-systems] currently supported by Confluent Platform.
- wget [https://www.gnu.org/software/wget/] to get the connector configuration file.

Download the Tutorial and Start ksqlDB

1. Clone the ksqlDB repository.

```
git clone https://github.com/confluentinc/ksql.git
cd ksql
```

2. Switch to the correct branch

Switch to the correct ksqlDB release branch.

```
git checkout 5.4.0-SNAPSHOT-post
```

3. Launch the tutorial in Docker

Navigate to the ksqlDB repository `docs/tutorials/` directory and launch the tutorial in Docker. Depending on your network speed, this may take up to 5-10 minutes.

```
cd docs/tutorials/
docker-compose up -d
```

4. Run the data generator tool

From two separate terminal windows, run the data generator tool to simulate "user" and "pageview" data:

```
docker run --network tutorials_default --rm --name datagen-pageviews \
    confluentinc/ksql-examples:0.6.0 \
    ksql-datagen \
        bootstrap-server=kafka:39092 \
        quickstart=pageviews \
        format=delimited \
        topic=pageviews \
        maxInterval=500
```

```
docker run --network tutorials_default --rm --name datagen-users \
    confluentinc/ksql-examples:0.6.0 \
    ksql-datagen \
        bootstrap-server=kafka:39092 \
```

```
        quickstart=users \
        format=json \
        topic=users \
        maxInterval=100
```

5. Start the ksqlDB CLI

From the host machine, start ksqlDB CLI:

```
docker run --network tutorials_default --rm --interactive --tty \
    confluentinc/ksqldb-cli:0.6.0 ksql \
    http://ksql-server:8088
```

Your output should resemble:

```
                  ===========================================
                  =       _              _ ____  ____       =
                  =      | | _____  __ _| |  _ \| __ )      =
                  =      | |/ / __|/ _` | | | | |  _ \      =
                  =      |   <\__ \ (_| | | |_| | |_) |     =
                  =      |_|\_\___/\__, |_|____/|____/      =
                  =                   |_|                   =
                  =  Event Streaming Database purpose-built =
                  =        for stream processing apps       =
                  ===========================================

Copyright 2017-2019 Confluent Inc.

CLI v0.6.0, Server v0.6.0 located at http://ksql-server:8088

Having trouble? Type 'help' (case-insensitive) for a rundown of how
things work!

ksql>
```

Inspect Kafka Topics By Using SHOW and PRINT Statements

ksqlDB enables inspecting Kafka topics and messages in real time.

- Use the SHOW TOPICS statement to list the available topics in the Kafka cluster.
- Use the PRINT statement to see a topic's messages as they arrive.

In the ksqlDB CLI, run the following statement:

```
SHOW TOPICS;
```

Your output should resemble:

```
 Kafka Topic        | Partitions | Partition Replicas
-----------------------------------------------------
 _confluent-metrics | 12         | 1
 _schemas           | 1          | 1
 pageviews          | 1          | 1
 users              | 1          | 1
-----------------------------------------------------
```

Inspect the `users` topic by using the PRINT statement:

```
PRINT 'users';
```

Your output should resemble:

```
    Format:JSON
{"ROWTIME":1540254230041,"ROWKEY":"User_1","registertime":1516754966866,

{"ROWTIME":1540254230081,"ROWKEY":"User_3","registertime":1491558386780,

{"ROWTIME":1540254230091,"ROWKEY":"User_7","registertime":1514374073235,

^C{"ROWTIME":1540254232442,"ROWKEY":"User_4","registertime":151003415137

    Topic printing ceased
```

Press Ctrl+C to stop printing messages.

Inspect the `pageviews` topic by using the PRINT statement:

```
PRINT 'pageviews';
```

Your output should resemble:

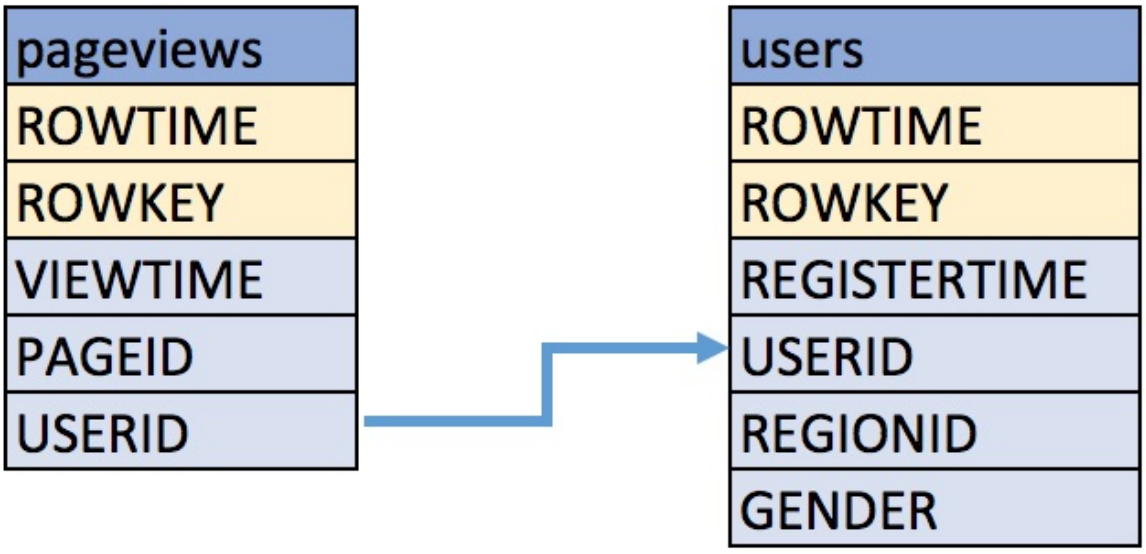
```
    Format:STRING
    10/23/18 12:24:03 AM UTC , 9461 , 1540254243183,User_9,Page_20
    10/23/18 12:24:03 AM UTC , 9471 , 1540254243617,User_7,Page_47
    10/23/18 12:24:03 AM UTC , 9481 , 1540254243888,User_4,Page_27
    ^C10/23/18 12:24:05 AM UTC , 9521 , 1540254245161,User_9,Page_62
    Topic printing ceased
    ksql>
```

Press Ctrl+C to stop printing messages.

For more information, see ksqlDB Syntax Reference [../../developer-guide/syntax-reference/].

Create a Stream and Table

These examples query messages from Kafka topics called `pageviews` and `users` using the following schemas:



1. Create a ksqlDB stream

Create a stream, named `pageviews_original`, from the `pageviews` Kafka topic, specifying the `value_format` of `DELIMITED`.

```
CREATE STREAM pageviews_original (viewtime bigint, userid varchar,
pageid varchar) WITH
(kafka_topic='pageviews', value_format='DELIMITED');
```

Your output should resemble:

```
 Message
---------------
 Stream created
---------------
```

Tip

You can run `DESCRIBE pageviews_original;` to see the schema for the stream. Notice that ksqlDB created two additional columns, named `ROWTIME`, which corresponds with the Kafka message timestamp, and `ROWKEY`, which corresponds with the Kafka message key.

2. Create a ksqlDB table

Create a table, named `users_original`, from the `users` topic, specifying the `value_format` of `JSON`.

```
CREATE TABLE users_original (registertime BIGINT, gender VARCHAR,
regionid VARCHAR, userid VARCHAR) WITH
(kafka_topic='users', value_format='JSON', key = 'userid');
```

Your output should resemble:

```
  Message
---------------
  Table created
---------------
```

Tip

You can run `DESCRIBE users_original;` to see the schema for the Table.

Optional: Show all streams and tables.

```
ksql> SHOW STREAMS;

Stream Name               | Kafka Topic               | Format
-----------------------------------------------------------------
PAGEVIEWS_ORIGINAL        | pageviews                 | DELIMITED

ksql> SHOW TABLES;

Table Name         | Kafka Topic       | Format    | Windowed
-------------------------------------------------------------
USERS_ORIGINAL     | users             | JSON      | false
```

Write Queries

These examples write queries using ksqlDB.

> **Note**
>
> By default ksqlDB reads the topics for streams and tables from the latest offset.

1. Create a query that returns data from a ksqlDB stream

Use `SELECT` to create a query that returns data from a STREAM. This query includes the `LIMIT` keyword to limit the number of rows returned in the query result. Note that exact data output may vary because of the randomness of the data generation.

```
SELECT pageid FROM pageviews_original LIMIT 3;
```

Your output should resemble:

```
Page_24
Page_73
Page_78
LIMIT reached
Query terminated
```

2. Create a persistent query

Create a persistent query by using the `CREATE STREAM` keywords to precede the `SELECT` statement. The results from this query are written to the `PAGEVIEWS_ENRICHED` Kafka topic. The following query enriches the `pageviews_original` STREAM by doing a `LEFT JOIN` with the `users_original` TABLE on the user ID.

```
CREATE STREAM pageviews_enriched AS
    SELECT users_original.userid AS userid, pageid, regionid, gender
    FROM pageviews_original
    LEFT JOIN users_original
    ON pageviews_original.userid = users_original.userid;
```

Your output should resemble:

```
  Message
----------------------------
  Stream created and running
----------------------------
```

> **Tip**
>
> You can run `DESCRIBE pageviews_enriched;` to describe the stream.

3. View query results

Use `SELECT` to view query results as they come in. To stop viewing the query results, press Ctrl-C. This stops printing to the console but it does not terminate the actual query. The query continues to run in the underlying ksqlDB application.

```
SELECT * FROM pageviews_enriched;
```

Your output should resemble:

```
1519746861328 | User_4 | User_4 | Page_58 | Region_5 | OTHER
1519746861794 | User_9 | User_9 | Page_94 | Region_9 | MALE
1519746862164 | User_1 | User_1 | Page_90 | Region_7 | FEMALE
^CQuery terminated
```

4. Create a filter query

Create a new persistent query where a condition limits the streams content, using `WHERE`. Results from this query are written to a Kafka topic called `PAGEVIEWS_FEMALE`.

```
CREATE STREAM pageviews_female AS
    SELECT * FROM pageviews_enriched
    WHERE gender = 'FEMALE';
```

Your output should resemble:

```
  Message
----------------------------
  Stream created and running
----------------------------
```

> **Tip**
>
> You can run `DESCRIBE pageviews_female;` to describe the stream.

5. Create a LIKE query

Create a new persistent query where another condition is met, using `LIKE`. Results from this query are written to the `pageviews_enriched_r8_r9` Kafka topic.

```
CREATE STREAM pageviews_female_like_89
    WITH (kafka_topic='pageviews_enriched_r8_r9') AS
SELECT * FROM pageviews_female
WHERE regionid LIKE '%_8' OR regionid LIKE '%_9';
```

Your output should resemble:

```
  Message
----------------------------
  Stream created and running
----------------------------
```

6. Count and group pageview events

Create a new persistent query that counts the pageviews for each region and gender combination in a tumbling window [https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#tumbling-time-windows] of 30 seconds when the count is greater than one. Results from this query are written to the `PAGEVIEWS_REGIONS` Kafka topic in the Avro format. ksqlDB registers the Avro schema with the configured Schema Registry when it writes the first message to the `PAGEVIEWS_REGIONS` topic.

```
CREATE TABLE pageviews_regions
    WITH (VALUE_FORMAT='avro') AS
SELECT gender, regionid , COUNT(*) AS numusers
FROM pageviews_enriched
    WINDOW TUMBLING (size 30 second)
GROUP BY gender, regionid
HAVING COUNT(*) > 1;
```

Your output should resemble:

```
  Message
---------------------------
  Table created and running
---------------------------
```

> **Tip**
>
> You can run `DESCRIBE pageviews_regions;` to describe the table.

7. View query results

View results from the previous queries by using the `SELECT` statement.

```
SELECT gender, regionid, numusers FROM pageviews_regions LIMIT 5;
```

Your output should resemble:

```
FEMALE | Region_6 | 3
FEMALE | Region_1 | 4
FEMALE | Region_9 | 6
MALE | Region_8 | 2
OTHER | Region_5 | 4
LIMIT reached
Query terminated
ksql>
```

8. View persistent queries

Show the running persistent queries:

```
SHOW QUERIES;
```

Your output should resemble:

```
Query ID                        | Kafka Topic              | Query
String
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-----------------------------------
CSAS_PAGEVIEWS_FEMALE_1         | PAGEVIEWS_FEMALE         | CREATE
STREAM pageviews_female AS       SELECT * FROM pageviews_enriched
WHERE gender = 'FEMALE';
CTAS_PAGEVIEWS_REGIONS_3        | PAGEVIEWS_REGIONS        | CREATE
TABLE pageviews_regions         WITH (VALUE_FORMAT='avro') AS
SELECT gender, regionid , COUNT(*) AS numusers       FROM
pageviews_enriched         WINDOW TUMBLING (size 30 second)
GROUP BY gender, regionid       HAVING COUNT(*) > 1;
CSAS_PAGEVIEWS_FEMALE_LIKE_89_2 | PAGEVIEWS_FEMALE_LIKE_89 | CREATE
STREAM pageviews_female_like_89         WITH
(kafka_topic='pageviews_enriched_r8_r9') AS       SELECT * FROM
pageviews_female       WHERE regionid LIKE '%_8' OR regionid LIKE
'%_9';
CSAS_PAGEVIEWS_ENRICHED_0       | PAGEVIEWS_ENRICHED       | CREATE
STREAM pageviews_enriched AS       SELECT users_original.userid AS
userid, pageid, regionid, gender       FROM pageviews_original
LEFT JOIN users_original         ON pageviews_original.userid =
users_original.userid;
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-----------------------------------
For detailed information on a Query run: EXPLAIN <Query ID>;
```

9. Examine query run-time metrics and details

Observe that information including the target Kafka topic is available, as well as throughput figures for the messages being processed.

```
DESCRIBE EXTENDED PAGEVIEWS_REGIONS;
```

Your output should resemble:

```
Name                 : PAGEVIEWS_REGIONS
Type                 : TABLE
Key field            : KSQL_INTERNAL_COL_0|+|KSQL_INTERNAL_COL_1
Key format           : STRING
Timestamp field      : Not set - using <ROWTIME>
Value format         : AVRO
Kafka topic          : PAGEVIEWS_REGIONS (partitions: 4, replication:
1)

Field    | Type
-------------------------------------
ROWTIME  | BIGINT           (system)
ROWKEY   | VARCHAR(STRING)  (system)
GENDER   | VARCHAR(STRING)
REGIONID | VARCHAR(STRING)
NUMUSERS | BIGINT
-------------------------------------

Queries that write into this TABLE
-----------------------------------
CTAS_PAGEVIEWS_REGIONS_3 : CREATE TABLE pageviews_regions         WITH
(value_format='avro') AS       SELECT gender, regionid , COUNT(*) AS
numusers       FROM pageviews_enriched         WINDOW TUMBLING (size
30 second)       GROUP BY gender, regionid       HAVING COUNT(*) > 1;

For query topology and execution plan please run: EXPLAIN <QueryId>

Local runtime statistics
------------------------
messages-per-sec:      3.06   total-messages:      1827     last-
message: 7/19/18 4:17:55 PM UTC
failed-messages:         0 failed-messages-per-sec:         0
last-failed:       n/a
(Statistics of the local KSQL server interaction with the Kafka topic
PAGEVIEWS_REGIONS)
ksql>
```

Use Nested Schemas (STRUCT) in ksqlDB

Struct support enables the modeling and access of nested data in Kafka topics, from both JSON and Avro.

1. Generate data

In this section, you use the `ksql-datagen` tool to create some sample data that includes a nested `address` field. Run this in a new window, and leave it running.

```
docker run --network tutorials_default --rm  \
  confluentinc/ksql-examples:0.6.0 \
  ksql-datagen \
      quickstart=orders \
      format=avro \
      topic=orders \
      bootstrap-server=kafka:39092 \
      schemaRegistryUrl=http://schema-registry:8081
```

2. Register a stream

From the ksqlDB command prompt, register the a stream on the `orders` topic:

```
CREATE STREAM ORDERS WITH (KAFKA_TOPIC='orders', VALUE_FORMAT='AVRO');
```

Your output should resemble:

```
 Message
----------------
 Stream created
----------------
```

3. Observe the stream's schema

Use the `DESCRIBE` function to observe the schema, which includes a `STRUCT`:

```
DESCRIBE ORDERS;
```

Your output should resemble:

```
Name                 : ORDERS
 Field      | Type
----------------------------------------------------------------------------
------------
 ROWTIME    | BIGINT           (system)
 ROWKEY     | VARCHAR(STRING)  (system)
 ORDERTIME  | BIGINT
 ORDERID    | INTEGER
 ITEMID     | VARCHAR(STRING)
 ORDERUNITS | DOUBLE
 ADDRESS    | STRUCT<CITY VARCHAR(STRING), STATE VARCHAR(STRING),
ZIPCODE BIGINT>
----------------------------------------------------------------------------
------------
For runtime statistics and query details run: DESCRIBE EXTENDED
<Stream,Table>;
ksql>
```

4. Access the struct data

Query the data by using the `->` notation to access the struct contents:

```
SELECT ORDERID, ADDRESS->CITY FROM ORDERS;
```

Your output should resemble:

```
0 | City_35
1 | City_21
2 | City_47
3 | City_57
4 | City_17
```

Press Ctrl+C to cancel the `SELECT` query.

Stream-Stream join

Using a stream-stream join, you can join two event streams on a common key. An example of this could be a stream of order events and a stream of shipment events. By joining these on the order key, you can see shipment information alongside the order.

1. Populate two source topics

In a separate console window, populate the `orders` and `shipments` topics by using the `kafkacat` utility:

```
docker run --interactive --rm --network tutorials_default \
  confluentinc/cp-kafkacat \
  kafkacat -b kafka:39092 \
          -t new_orders \
          -K: \
          -P <<EOF
1:{"order_id":1,"total_amount":10.50,"customer_name":"Bob Smith"}
2:{"order_id":2,"total_amount":3.32,"customer_name":"Sarah Black"}
3:{"order_id":3,"total_amount":21.00,"customer_name":"Emma Turner"}
EOF
```

```
docker run --interactive --rm --network tutorials_default \
  confluentinc/cp-kafkacat \
  kafkacat -b kafka:39092 \
          -t shipments \
          -K: \
          -P <<EOF
1:{"order_id":1,"shipment_id":42,"warehouse":"Nashville"}
3:{"order_id":3,"shipment_id":43,"warehouse":"Palo Alto"}
EOF
```

2. Register two streams

In the ksqlDB CLI, register both topics as ksqlDB streams:

```
CREATE STREAM NEW_ORDERS (ORDER_ID INT, TOTAL_AMOUNT DOUBLE,
CUSTOMER_NAME VARCHAR)
WITH (KAFKA_TOPIC='new_orders', VALUE_FORMAT='JSON');

CREATE STREAM SHIPMENTS (ORDER_ID INT, SHIPMENT_ID INT, WAREHOUSE
VARCHAR)
WITH (KAFKA_TOPIC='shipments', VALUE_FORMAT='JSON');
```

After both `CREATE STREAM` statements, your output should resemble:

```
 Message
----------------
 Stream created
----------------
```

3. Set the auto.offset.reset property

Run the following statement to tell ksqlDB to read from the beginning of all topics:

```
SET 'auto.offset.reset' = 'earliest';
```

> **Tip**
>
> You can skip this step if you've already run it within your current ksqlDB CLI session.

4. Examine streams for events

Query the streams to confirm that events are present in the topics.

For the `NEW_ORDERS` stream, run:

```
SELECT ORDER_ID, TOTAL_AMOUNT, CUSTOMER_NAME FROM NEW_ORDERS LIMIT 3;
```

Your output should resemble:

```
1 | 10.5 | Bob Smith
2 | 3.32 | Sarah Black
3 | 21.0 | Emma Turner
```

For the `SHIPMENTS` stream, run:

```
SELECT ORDER_ID, SHIPMENT_ID, WAREHOUSE FROM SHIPMENTS LIMIT 2;
```

Your output should resemble:

```
1 | 42 | Nashville
3 | 43 | Palo Alto
```

5. Join the streams

Run the following query, which will show orders with associated shipments, based on a join window of 1 hour.

```
SELECT O.ORDER_ID, O.TOTAL_AMOUNT, O.CUSTOMER_NAME,
S.SHIPMENT_ID, S.WAREHOUSE
FROM NEW_ORDERS O
INNER JOIN SHIPMENTS S
  WITHIN 1 HOURS
  ON O.ORDER_ID = S.ORDER_ID;
```

Your output should resemble:

```
1 | 10.5 | Bob Smith | 42 | Nashville
3 | 21.0 | Emma Turner | 43 | Palo Alto
```

Messages with `ORDER_ID=2` have no corresponding `SHIPMENT_ID` or `WAREHOUSE`. This is because there's no corresponding row on the `SHIPMENTS` stream within the time window specified.

Press Ctrl+C to cancel the `SELECT` query and return to the prompt.

Table-Table join

Using a table-table join, it's possible to join two tables of on a common key. ksqlDB tables provide the latest *value* for a given *key*. They can only be joined on the *key*, and one-to-many (1:N) joins are not supported in the current semantic model.

In this example, location data about a warehouse from one system is enriched with data about the size of the warehouse from another.

1. Populate two source topics

In a separate console window, populate the two topics by using the `kafkacat` utility:

```
docker run --interactive --rm --network tutorials_default \
  confluentinc/cp-kafkacat \
  kafkacat -b kafka:39092 \
          -t warehouse_location \
          -K: \
          -P <<EOF
1:{"warehouse_id":1,"city":"Leeds","country":"UK"}
2:{"warehouse_id":2,"city":"Sheffield","country":"UK"}
3:{"warehouse_id":3,"city":"Berlin","country":"Germany"}
EOF
```

```
docker run --interactive --rm --network tutorials_default \
  confluentinc/cp-kafkacat \
  kafkacat -b kafka:39092 \
          -t warehouse_size \
          -K: \
          -P <<EOF
1:{"warehouse_id":1,"square_footage":16000}
2:{"warehouse_id":2,"square_footage":42000}
3:{"warehouse_id":3,"square_footage":94000}
EOF
```

2. Register two tables

In the ksqlDB CLI, register both topics as ksqlDB tables:

```
CREATE TABLE WAREHOUSE_LOCATION (WAREHOUSE_ID INT, CITY VARCHAR,
COUNTRY VARCHAR)
WITH (KAFKA_TOPIC='warehouse_location',
      VALUE_FORMAT='JSON',
      KEY='WAREHOUSE_ID');

CREATE TABLE WAREHOUSE_SIZE (WAREHOUSE_ID INT, SQUARE_FOOTAGE DOUBLE)
WITH (KAFKA_TOPIC='warehouse_size',
      VALUE_FORMAT='JSON',
      KEY='WAREHOUSE_ID');
```

After both `CREATE TABLE` statements, your output should resemble:

```
 Message
---------------
 Table created
---------------
```

3. Examine tables for keys

Check both tables that the message key (`ROWKEY`) matches the declared key (`WAREHOUSE_ID`). The output should show that they are equal. If they aren't, the join won't succeed or behave as expected.

Inspect the WAREHOUSE_LOCATION table:

```
SELECT ROWKEY, WAREHOUSE_ID FROM WAREHOUSE_LOCATION LIMIT 3;
```

Your output should resemble:

```
1 | 1
2 | 2
3 | 3
Limit Reached
Query terminated
```

Inspect the WAREHOUSE_SIZE table:

```
SELECT ROWKEY, WAREHOUSE_ID FROM WAREHOUSE_SIZE LIMIT 3;
```

Your output should resemble:

```
1 | 1
2 | 2
3 | 3
Limit Reached
Query terminated
```

4. Join the tables

Now join the two tables:

```
SELECT WL.WAREHOUSE_ID, WL.CITY, WL.COUNTRY, WS.SQUARE_FOOTAGE
FROM WAREHOUSE_LOCATION WL
  LEFT JOIN WAREHOUSE_SIZE WS
    ON WL.WAREHOUSE_ID=WS.WAREHOUSE_ID
LIMIT 3;
```

Your output should resemble:

```
1 | Leeds | UK | 16000.0
2 | Sheffield | UK | 42000.0
3 | Berlin | Germany | 94000.0
Limit Reached
Query terminated
```

INSERT INTO

The `INSERT INTO` syntax can be used to merge the contents of multiple streams. An example of this could be where the same event type is coming from different sources.

1. Generate data

Run two datagen processes, each writing to a different topic, simulating order data arriving from a local installation vs from a third-party:

```
docker run --network tutorials_default --rm  --name datagen-orders-
local \
  confluentinc/ksql-examples:0.6.0 \
  ksql-datagen \
      quickstart=orders \
      format=avro \
      topic=orders_local \
      bootstrap-server=kafka:39092 \
      schemaRegistryUrl=http://schema-registry:8081
```

```
docker run --network tutorials_default --rm --name datagen-
orders_3rdparty \
  confluentinc/ksql-examples:0.6.0 \
  ksql-datagen \
      quickstart=orders \
      format=avro \
      topic=orders_3rdparty \
      bootstrap-server=kafka:39092 \
      schemaRegistryUrl=http://schema-registry:8081
```

2. Register streams

In ksqlDB, register the source topic for each:

```
CREATE STREAM ORDERS_SRC_LOCAL
  WITH (KAFKA_TOPIC='orders_local', VALUE_FORMAT='AVRO');

CREATE STREAM ORDERS_SRC_3RDPARTY
  WITH (KAFKA_TOPIC='orders_3rdparty', VALUE_FORMAT='AVRO');
```

After each `CREATE STREAM` statement you should get the message:

```
 Message
----------------
```

```
 Stream created
----------------
```

3. Create a persistent query

Create the output stream, using the standard `CREATE STREAM … AS` syntax. Because multiple sources of data are being joined into a common target, it is useful to add in lineage information. This can be done by simply including it as part of the `SELECT`:

```
CREATE STREAM ALL_ORDERS AS SELECT 'LOCAL' AS SRC, * FROM
ORDERS_SRC_LOCAL;
```

Your output should resemble:

```
 Message
----------------------------
 Stream created and running
----------------------------
```

4. Examine the stream's schema

Use the `DESCRIBE` command to observe the schema of the target stream.

```
DESCRIBE ALL_ORDERS;
```

Your output should resemble:

```
Name                 : ALL_ORDERS
 Field      | Type
-----------------------------------------------------------------------
------------
 ROWTIME    | BIGINT           (system)
 ROWKEY     | VARCHAR(STRING)  (system)
 SRC        | VARCHAR(STRING)
 ORDERTIME  | BIGINT
 ORDERID    | INTEGER
 ITEMID     | VARCHAR(STRING)
 ORDERUNITS | DOUBLE
```

```
 ADDRESS    | STRUCT<CITY VARCHAR(STRING), STATE VARCHAR(STRING),
ZIPCODE BIGINT>
--------------------------------------------------------------------------
------------
For runtime statistics and query details run: DESCRIBE EXTENDED
<Stream,Table>;
```

5. Insert another stream

Add a stream of 3rd-party orders into the existing output stream:

```
INSERT INTO ALL_ORDERS SELECT '3RD PARTY' AS SRC, * FROM
ORDERS_SRC_3RDPARTY;
```

Your output should resemble:

```
 Message
-------------------------------
 Insert Into query is running.
-------------------------------
```

6. Query the output stream

Query the output stream to verify that data from each source is being written to it:

```
SELECT * FROM ALL_ORDERS;
```

Your output should resemble:

```
1531736084879 | 1802 | 3RD PARTY | 1508543844870 | 1802 | Item_427 |
5.003326679575532 | {CITY=City_27, STATE=State_63, ZIPCODE=12589}
1531736085016 | 1836 | LOCAL | 1489112050820 | 1836 | Item_224 |
9.561788841477156 | {CITY=City_67, STATE=State_99, ZIPCODE=28638}
1531736085118 | 1803 | 3RD PARTY | 1516295084125 | 1803 | Item_208 |
7.984495994658404 | {CITY=City_13, STATE=State_56, ZIPCODE=23417}
1531736085222 | 1804 | 3RD PARTY | 1503734687976 | 1804 | Item_498 |
4.8212828530483876 | {CITY=City_42, STATE=State_45, ZIPCODE=87842}
1531736085444 | 1837 | LOCAL | 1511189492298 | 1837 | Item_183 |
```

```
1.3867306505950954 | {CITY=City_28, STATE=State_86, ZIPCODE=14742}
1531736085531 | 1838 | LOCAL | 1497601536360 | 1838 | Item_945 |
4.825111590185673 | {CITY=City_78, STATE=State_13, ZIPCODE=59763}
…
```

Events from both source topics are present, denoted by `LOCAL` and `3RD PARTY` respectively.

Press Ctrl+C to cancel the `SELECT` query and return to the ksqlDB prompt.

7. View the queries

You can view the two queries that are running using `SHOW QUERIES`:

```
SHOW QUERIES;
```

Your output should resemble:

```
 Query ID          | Kafka Topic | Query String
------------------------------------------------------------------------
-------------------------------------------
 CSAS_ALL_ORDERS_0 | ALL_ORDERS  | CREATE STREAM ALL_ORDERS AS SELECT
'LOCAL' AS SRC, * FROM ORDERS_SRC_LOCAL;
 InsertQuery_1     | ALL_ORDERS  | INSERT INTO ALL_ORDERS SELECT '3RD
PARTY' AS SRC, * FROM ORDERS_SRC_3RDPARTY;
------------------------------------------------------------------------
-------------------------------------------
```

Terminate and Exit

ksqlDB

Important

Persisted queries run continuously as ksqlDB applications until they're terminated manually. Exiting the ksqlDB CLI doesn't terminate persistent queries.

From the output of `SHOW QUERIES;` identify a query ID you would like to terminate. For example, if you wish to terminate query ID `CTAS_PAGEVIEWS_REGIONS`:

```
TERMINATE CTAS_PAGEVIEWS_REGIONS;
```

> **Tip**
>
> The actual name of the query running may vary; refer to the output of `SHOW QUERIES;`.

Run the `exit` command to leave the ksqlDB CLI.

```
ksql> exit
Exiting ksqlDB.
```

Docker

To stop all data generator containers, run the following:

```
docker ps|grep ksql-datagen|awk '{print $1}'|xargs -Ifoo docker stop
foo
```

If you're running Kafka using Docker Compose, you can stop it and remove the containers and their data with this command.

```
cd docs/tutorials/
docker-compose down
```

Produce extra topic data and verify your environment

The following instructions aren't required to run the quick start. They're optional steps to produce extra topic data and verify the environment.

Produce more topic data

The Compose file automatically runs a data generator that continuously produces data to two Kafka topics, `pageviews` and `users`. No further action is required if you want to use just the data available. You can return to the main ksqlDB quick start [#create-a-stream-and-table] to start querying the data in these two topics.

However, if you want to produce additional data, you can use the following methods.

Produce Kafka data with the Kafka command line `kafka-console-producer`. The following example generates data with a value in DELIMITED format.

```
docker-compose exec kafka kafka-console-producer \
                            --topic t1 \
                            --broker-list kafka:39092  \
                            --property parse.key=true \
                            --property key.separator=:
```

Your data input should resemble:

```
key1:v1,v2,v3
key2:v4,v5,v6
key3:v7,v8,v9
key1:v10,v11,v12
```

Produce Kafka data with the Kafka command line `kafka-console-producer`. The following example generates data with a value in JSON format.

```
docker-compose exec kafka kafka-console-producer \
                            --topic t2 \
                            --broker-list kafka:39092  \
                            --property parse.key=true \
                            --property key.separator=:
```

Your data input should resemble:

```
key1:{"id":"key1","col1":"v1","col2":"v2","col3":"v3"}
key2:{"id":"key2","col1":"v4","col2":"v5","col3":"v6"}
key3:{"id":"key3","col1":"v7","col2":"v8","col3":"v9"}
key1:{"id":"key1","col1":"v10","col2":"v11","col3":"v12"}
```

You can also use `kafkacat` from Docker, as demonstrated in the previous examples.

Verify your environment

The following steps are optional verification steps to ensure your environment is properly setup.

Verify that six Docker containers were created.

```
docker-compose ps
```

Your output should resemble:

```
        Name                       Command            State
Ports
------------------------------------------------------------------------
------------------------------
tutorials_kafka_1             /etc/confluent/docker/run   Up
0.0.0.0:39092->39092/tcp, 9092/tcp
tutorials_ksql-server_1       /etc/confluent/docker/run   Up
8088/tcp
tutorials_schema-registry_1   /etc/confluent/docker/run   Up
8081/tcp
tutorials_zookeeper_1         /etc/confluent/docker/run   Up
2181/tcp, 2888/tcp, 3888/tcp
```

Take note of the `Up` state.

Earlier steps in this quickstart started two data generators that pre-populate two topics, named `pageviews` and `users`, with mock data. Verify that the data generator created these topics, including `pageviews` and `users`.

```
docker-compose exec kafka kafka-topics --zookeeper zookeeper:32181 --
list
```

Your output should resemble this.

```
_confluent-metrics
_schemas
pageviews
users
```

Use the `kafka-console-consumer` to view a few messages from each topic. The topic `pageviews` has a key that is a mock time stamp and a value that is in `DELIMITED` format. The topic `users` has a key that is the user ID and a value that is in `Json` format.

```
docker-compose exec kafka kafka-console-consumer \
                            --topic pageviews \
                            --bootstrap-server kafka:39092 \
                            --from-beginning \
                            --max-messages 3 \
                            --property print.key=true
```

Your output should resemble:

```
1491040409254    1491040409254,User_5,Page_70
1488611895904    1488611895904,User_8,Page_76
1504052725192    1504052725192,User_8,Page_92
```

```
docker-compose exec kafka kafka-console-consumer \
                            --topic users \
                            --bootstrap-server kafka:39092 \
                            --from-beginning \
                            --max-messages 3 \
                            --property print.key=true
```

Your output should resemble:

```
User_2
{"registertime":1509789307038,"gender":"FEMALE","regionid":"Region_1","u

User_6
{"registertime":1498248577697,"gender":"OTHER","regionid":"Region_8","us

User_8
```

```
{"registertime":1494834474504,"gender":"MALE","regionid":"Region_5","use
```

Next Steps

- Try the end-to-end Clickstream Analysis demo [../clickstream-docker/], which shows how to build an application that performs real-time user analytics.

Page last revised on: 2019-12-17

Last update: 2019-12-17